Contents

3.2 Immutable Data Structures

To support scalable execution a concurrent system should support speculation without increasing the effective memory bandwidth of the program. A solution should facilitate concurrent access to shared data while requiring that values are written to main memory only once. Immutable data is necessarily written to main memory only once so we propose that Immutable Data Structures can act as repositories of both speculative and shared state. Immutable Data Structures have not previously been considered in the context of concurrent execution so support for them must be developed before this proposal can be evaluated.

The problem is to find a mechanism for maintaining speculative and shared state in memory. The solution should support the isolation of speculative state and the atomic transformation of speculative state into shared state. It should also support simultaneous access to shared state and require that data values be written to main memory once only.

This section identifies Immutable Data Structures as candidate repositories of shared state in concurrent systems and examines techniques for maintaining both speculative and shared state in Immutable Data Structures.

3.2.1 Supporting Speculation

To support speculation a mechanism to isolate speculative state and permit its atomic transformation into shared state is required. This mechanism should afford scalable concurrency without increasing the effective memory bandwidth of the program.

The mechanism should support the isolation of speculative state from other functions executing concurrently. Only the process that wrote the state speculatively should be able to observe it. The mechanism should also ensure that a function observes a consistent view of shared state. Consistency criteria must be met at the point speculative state becomes shared state.

The mechanism should support the atomic transformation of speculative state into shared state. In a Chip Multi-Processor the only mechanism for performing an atomic action is an atomic instruction, so the transformation of speculative state into shared state must be implemented by an atomic instruction.

Typically, atomic instructions act on only one word in memory. The atomic transformation of isolated multi-word values into shared values can be achieved by atomically updating a reference to those values instead of the values themselves. To enable atomic transformation, to shared state, speculative state should be identified by a single reference and this reference should be modified by an atomic instruction.

An atomic instruction typically implements a memory barrier to ensure that any memory writes, buffered by the processor, are completed and that caches are coherent during the execution of the atomic instruction. The memory barrier ensures that the speculative state identified by the reference appears to be atomically transformed into shared state.

3.2.2 Immutable Data Structures

Immutable Data Structures provide a solution to the problem of maintaining both speculative and shared state. Paths within an Immutable Data Structure can be isolated until the mutable reference to the data structure is modified by an atomic instruction so functions acting on Immutable Data Structure can benefit from isolation and atomicity provided by the structures themselves.

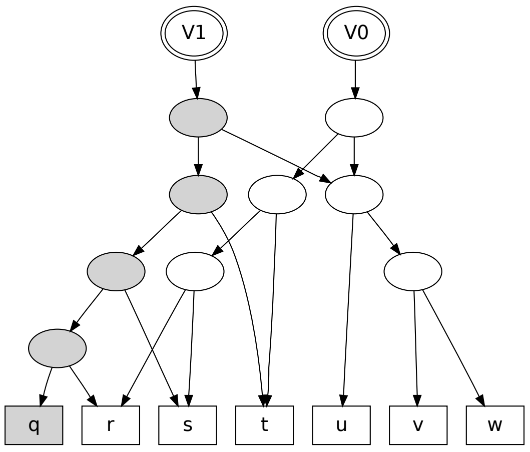
Figure 3.1 illustrates the insertion and removal of an element in an immutable binary tree. The functions cause a new path to be created within the data structure but do not change any of the existing values. A version of a data structure is identified by a mutable reference. The data structure does not change per se. Instead, a new version is created by copying data and modifying the reference.

3.2.3 Immutability and Concurrency

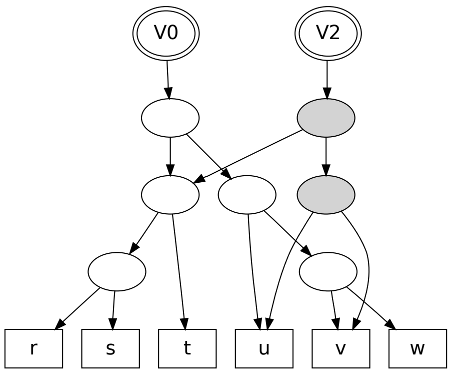
In this section we describe how certainty that shared data within an Immutable Data Structure is immutable enables a program to access it concurrently.

Immutable Data Structures provide a medium for maintaining immutable shared state within the data structure itself. Immutable Data Structures also provide a medium for maintaining isolated speculative state, in the form of the values written by an access function. The mutable reference to the data structure is modified by an atomic instruction and this causes the speculative state, created in isolation by the access function, to be transformed atomically into shared state.

Concurrent accesses to mutable data structures must be coordinated for two reasons. Firstly to protect the integrity of the data structure itself and secondly



(a)



(b)

Figure 3.1: **Insertion and deletion from an immutable binary tree.** The shaded vertices represent the path created by the operation. An ellipse with a double border represents a mutable reference to a version of the Immutable Data Structure. Version V0 of the immutable binary tree contains the elements $\{r, s, t, u, v, w\}$.
(a) Insertion of an element q into an immutable binary tree creates version V1 containing the elements $\{q, r, s, t, u, v, w\}$.
(b) Removal of the element w from an immutable binary tree creates version V2 containing the elements $\{r, s, t, u, v\}$.

to ensure the correct semantic order of operation. An Immutable Data Structure distinguishes between the structural consistency criteria of the data structure and the semantic consistency criteria of the application data. However, Immutable Data Structures do not offer a mechanism for ensuring the correct ordering of the effects of concurrent operations. A mechanism to ensure this ordering is presented in subsequent chapters.

Bibliography